EECS 432 Advanced Computer Vision

Project Title: Self-Driving Turtlebot3 Based on Lane Lines Keeping and Traffic Sign Recognition

Member: Peng Peng

Project Description

Goal

My project goal is to build a elementary self-driving robot, which can follow the lane lines and recognize the traffic signs it face to, and thus do corresponding performance.

Problem

For this project, we face two main difficulty: how to implement dynamics control of robot based on the road situation, and how to make the robot grasp all the traffic sign on the road robustly.

Project Architecture

Advanced Lane Finding:

1. Camera Calibration and Distortion Correction

2. Thresholded binary image based on color transforms and gradients

3. Perspective Transform(birds-eye view

4. Identifying the lane (a) Histogram (b) Sliding Window Search

5. Computing lane’s curvature and center deviation (a) Ployfit the lane line can make calculation (b) Inverse Transform

6. PID controller based on curvature and center deviation

Traffic Sign Detection

1. Detecting Feature Choice

2. Building and Training Classifier

3. Multi-scale Sliding Windows search

4. Multiple Detections & False Positives

Traffic Sign Classifier

1. Dataset choice and its preprocess

2. Deep CNN Architecture

3. Dropout

4. Data Augment

Part 1. Advanced Lane Finding

Ideas Flow:

I decided to implement a PID controller to make the robot follow the lane lines. To do this, we need to know the robot position relative to the road center. Moreover, we need the curvature of the lane lines to help us control the robot's steer angle. To get these road data, I decided to get the "birds-eye view" of the road, thus I implement a perspective transform. To find the road area we want to transform, we need detect the lane lines. So I choosed the combination of color and gradient threshold to help me select the lane lines out.

1.Camera Calibration

I collected 20 images of chessboard from varying angle and distance served as camera calibration input, feed them into *cv2.findChessboardCorners* to get object points and image points, then feed the result into cv2.calibrateCamera to get the Camera Matrix. Here is the result after distortion correction.

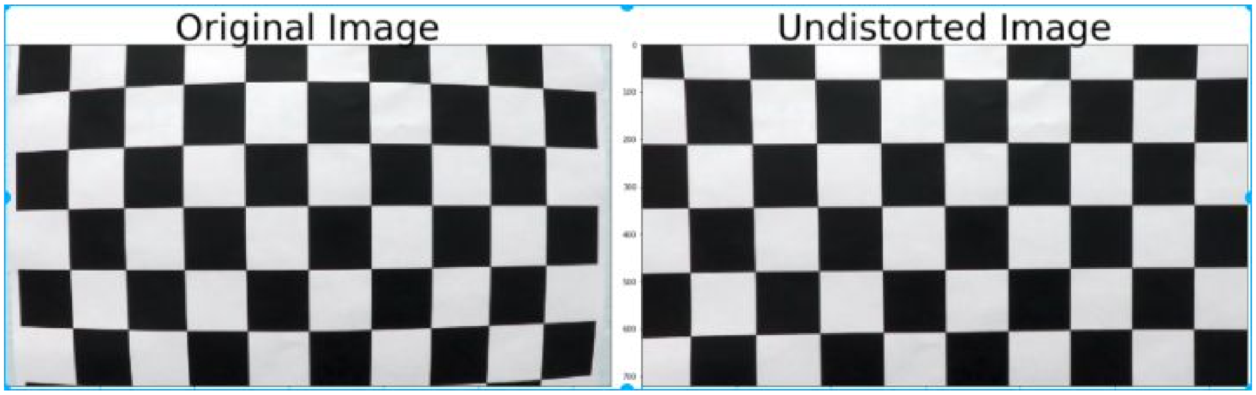
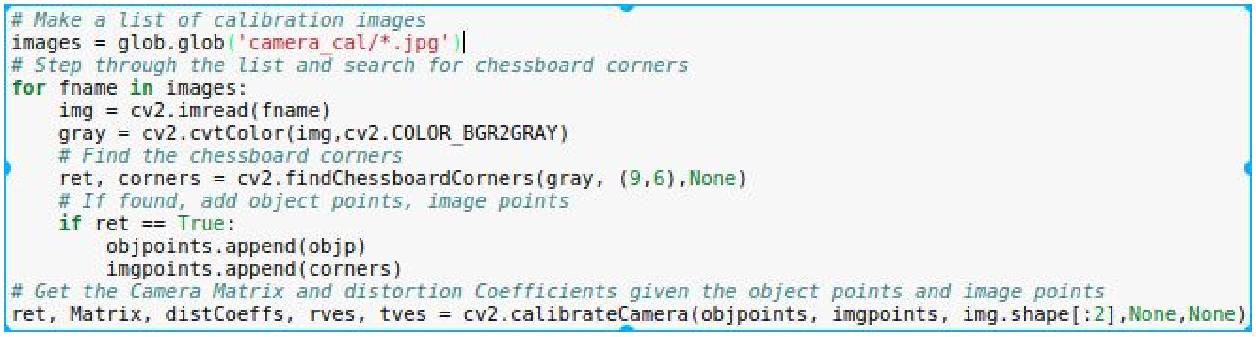


Figure 1. Camera Calibration



```
# Make a list of calibration images
images = glob.glob('camera_cal/*.jpg')
# Step through the list and search for chessboard corners
for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (9,6),None)
    # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)
# Get the Camera Matrix and distortion Coefficients given the object points and image points
ret, Matrix, distCoeffs, rves, tves = cv2.calibrateCamera(objpoints, imgpoints, img.shape[:2],None,None)
```

2. Thresholded binary image based on color transforms and gradients

The color of two lane lines are black and orange respectively. So I use low L channel in HLS to find black lane, and low B channel in RGB to find orange lane. In addition, to make the land finding mechanism more robust, I implement gradient threshold to filter out nearly vertical or horizontal lines, which are not likely to be lane lines. Here is the thresholded image.

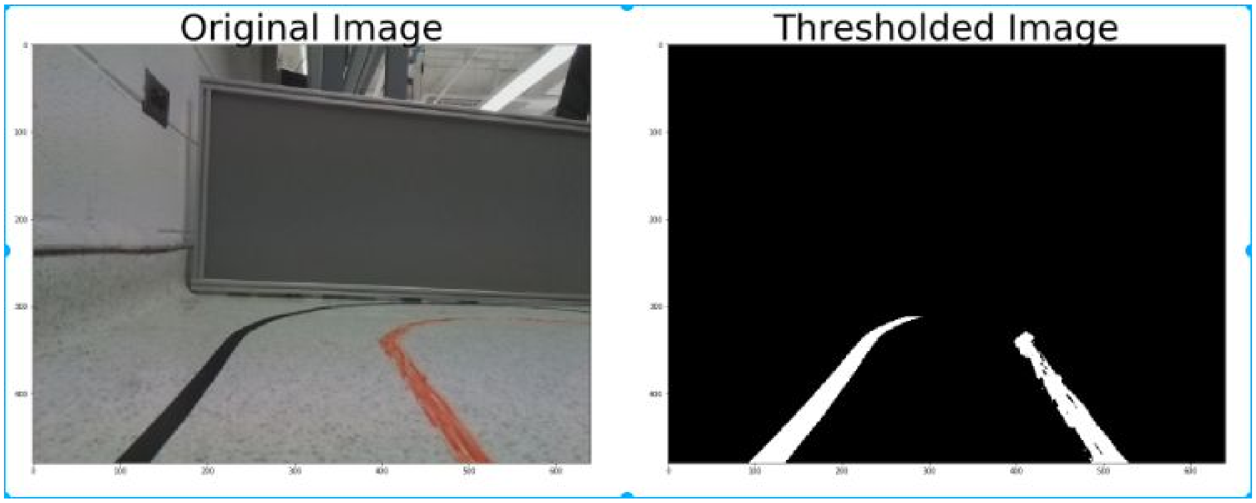
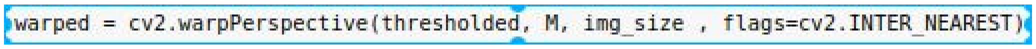


Figure 2. Thresholded Binary Image

3. Perspective Transform(birds-eye view)

Then I implement birds-eye-view transform by choose the transform boundary manually(by *cv2.warpPerspective*).



```
warped = cv2.warpPerspective(thresholded, M, img_size , flags=cv2.INTER_NEAREST)
```

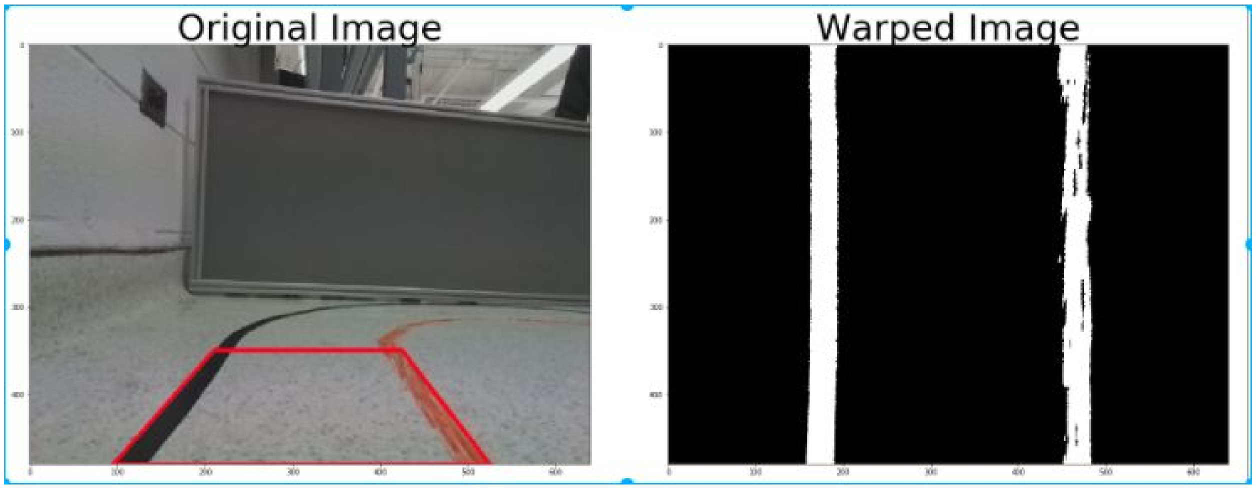


Figure 3. Camera Calibration

Thus we can see the robot deviation and lane curvature by warped image directly!

In addition, I also get the conversion relationship between distance in real world and pixels in warped image by measuring the size of the transform area.

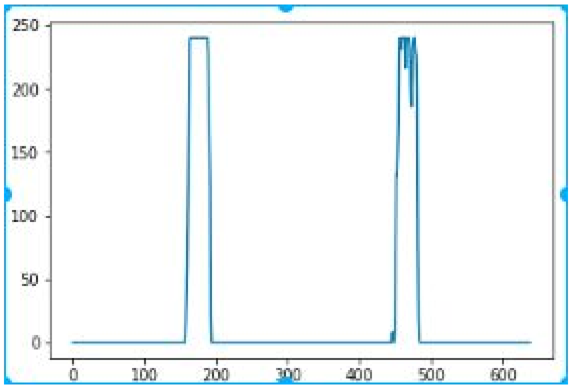
ym_per_pixel = 22.5/32000; **xm_per_pixel** = 17.5/32000

Which can be used in the following curvature calculation.

4. Identifying the lane

a.Line Finding Methods: Peaks in a Histogram

After applying calibration, thresholding and perspective transform, to decide explicitly which pixels are part of the left lines or right lines, I take a histogram along all the columns in the lower half of the image. Then the two most prominent peaks in the histogram will be good indicators of the x-position of lane lines.



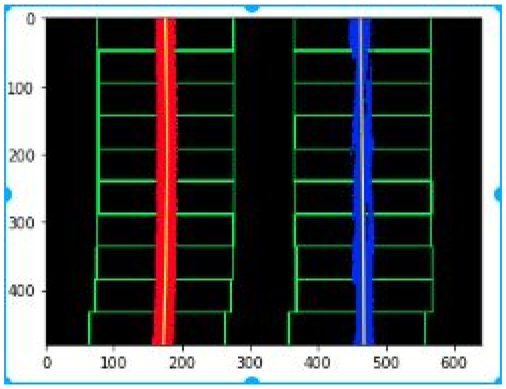


Figure 4. Histogram of Thresholded **Figure 5. Sliding Window Search of Lane Lines**

b. sliding window search

I use that as a starting point for where to search for the lines. From that point, I then use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

Moreover, after the first several video frame, once we know where the previous lines are, we can search the line for the next frame in a margin around the previous line position, instead of a blind search again.

5. Measuring Curvature

a. Fit a second order polynomial curve

Before we measuring the curvature, we firstly need to use a second(or third, if multiple consecutive turn) order polynomial to fit the points set we get above.

```
# Fit a second order polynomial to each
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

b. Curvature Calculation

I get the curvature by pixel form following formula:

$$f(y) = Ay^2 + By + C \qquad R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|} \qquad R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

Then according to the pixels to real world factors **ym_per_pixel** = 22.5/32000; **xm_per_pixel** = 17.5/32000, we can get the lane curvature in real world.

In addition, we get the robot center deviation by two lanes’ x coordinates.

C. After the calculation, I implemented an inverse perspective transform to mark the lane lines area.

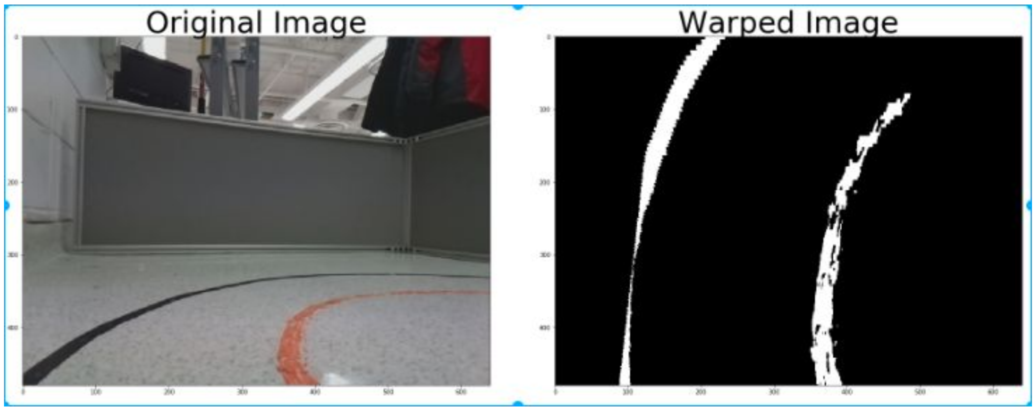
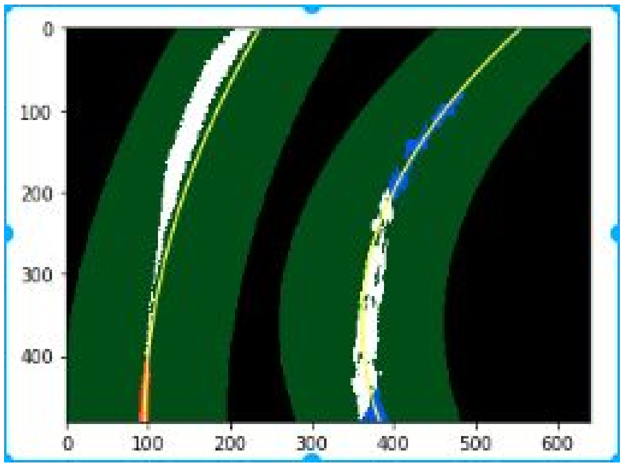


Figure 6. Original and Warped Image for Curvature Measurement



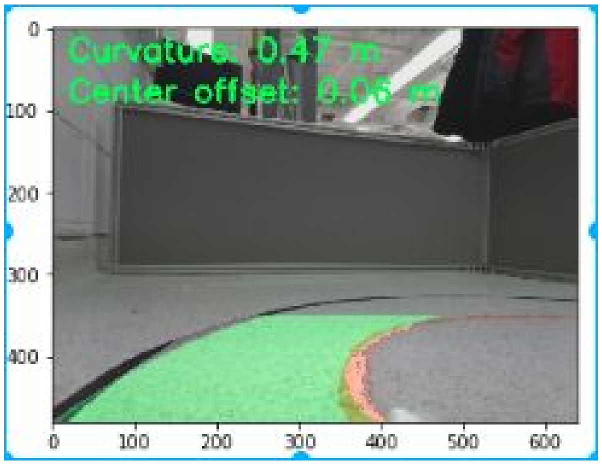


Figure 7. FIt for Second Order Polynomial

Figure 8. Measurement Result & Inverse Transform

6. PID Controller

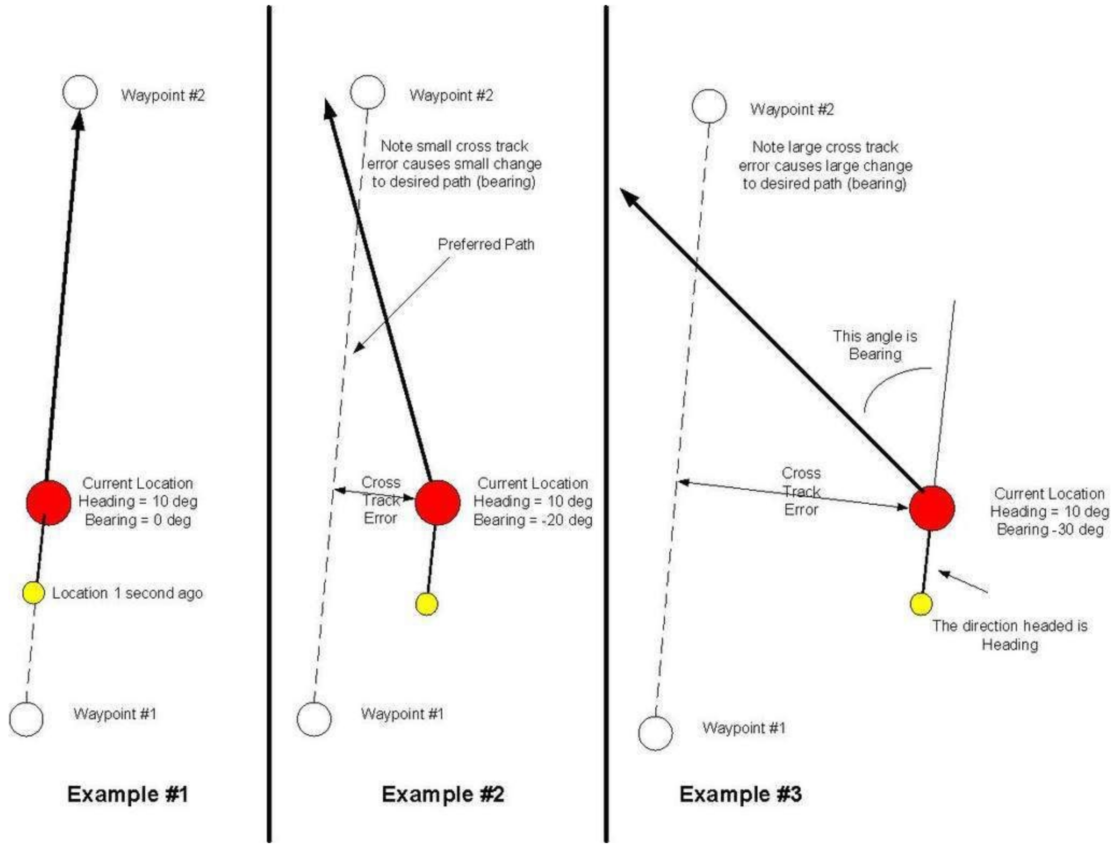


Figure 9. Cross-track Error Definition

The cross-track error's(CTE) definition just as above show. In this project, the CTE is exactly the same as center offset we figure out. We'll make CTE and R_Curve served as the input of PID controller. It's output would be the robot angular velocity.

The following part is generally a pre-process for the traffic sign classifier. It is hard for me to implement the CNN on the whole image directly to classifier the traffic sign. So I separate the traffic sign recognition into two parts. The first is to get the region of image which contains a traffic sign by traditional computer vision methods, and we withhold to classifier it. The second is to feed this area into a deep CNN to classifier it.

Part 2. Traffic Sign Detection

1. Detecting Feature Choice

To select the traffic sign out from the lab background, we need to find some image feature which is exclusive to traffic sign. The color components is not doubtedly could be a good indicator, cauze traffic sign usually is a combination of white and blue or red. We can represent by color histogram. Another indicator is the special structure of traffic sign, and we can use histogram of gradient to represent it.

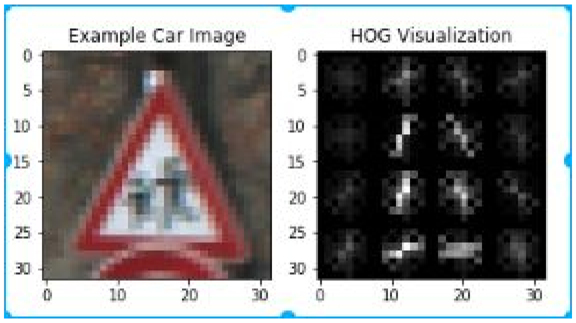


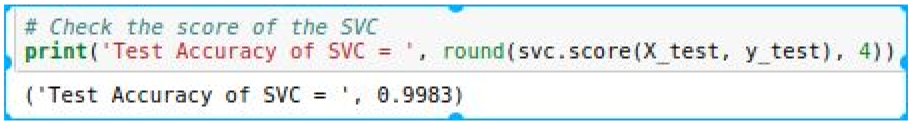
Figure 10. HOG Features

Finally, I choose the combination of color and HOG features to feed the detector.

2. Building and Training Classifier

As for the classifier, I choose a linear SVM to classify the traffic sign. Specifically, I'll train the model from the German Traffic Sign Dataset. It contains more than 20,000 traffic signs image in 32*32*3 size, which label as 1(traffic signs). In addition, I also collect more than 20,000 images in 32*32*3 in our robotics lab environment, which label as 0. I keep these two training dataset's number of image balanced, to make the training result do not biased.

I split 20% of training data as the test data, the traffic sign classifier final performance is as following:



```
# Check the score of the SVC
print('Test Accuracy of SVC = ', round(svc.score(X_test, y_test), 4))
('Test Accuracy of SVC = ', 0.9983)
```

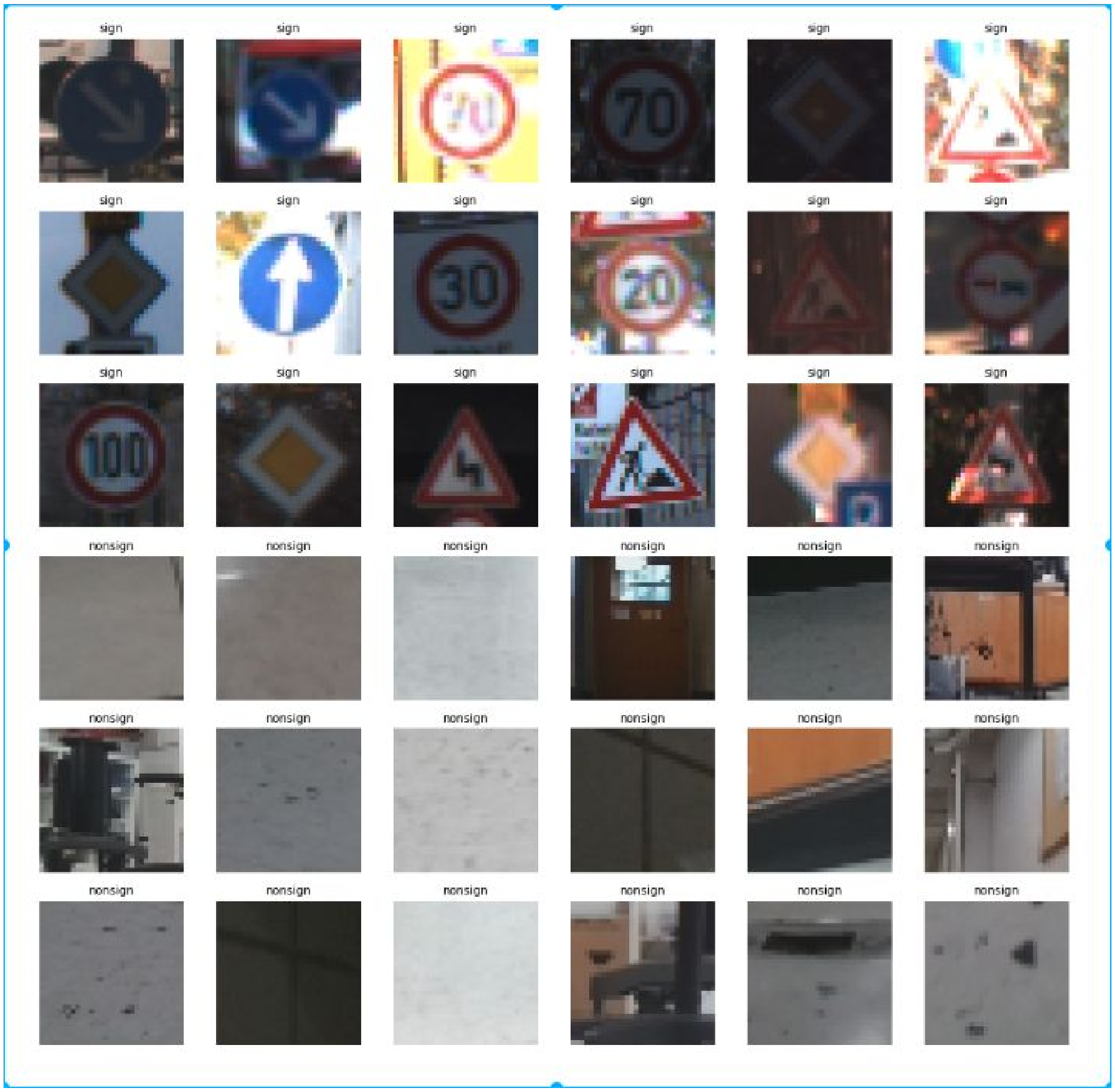


Figure 11. Dataset for Traffic Sign Detection(Sign & non-sign)

3. Multi-scale Sliding Windows search

Now that we get the classifier, what we're going to is to implement Multi-scale sliding windows to sample the image's subregion features, feed them into classifier and make a prediction that whether the subregion contain a traffic sign or not.

Before we implement a brute sliding windows search, here are some tricks about the search windows:

(a) Considering the robots camera view, which the ground occupy the lower half of the image, we know that traffic sign put on the road would not appear at the higher part of the image. Thus we only implement sliding window search at the lower part of the image.

(b) Considering traffic signs which are far away from the robot(which are smaller and locate at the center of y axis of the image), we only implement small sliding window at the half of y axis position.

Finally, the sliding window we implement just as follow:

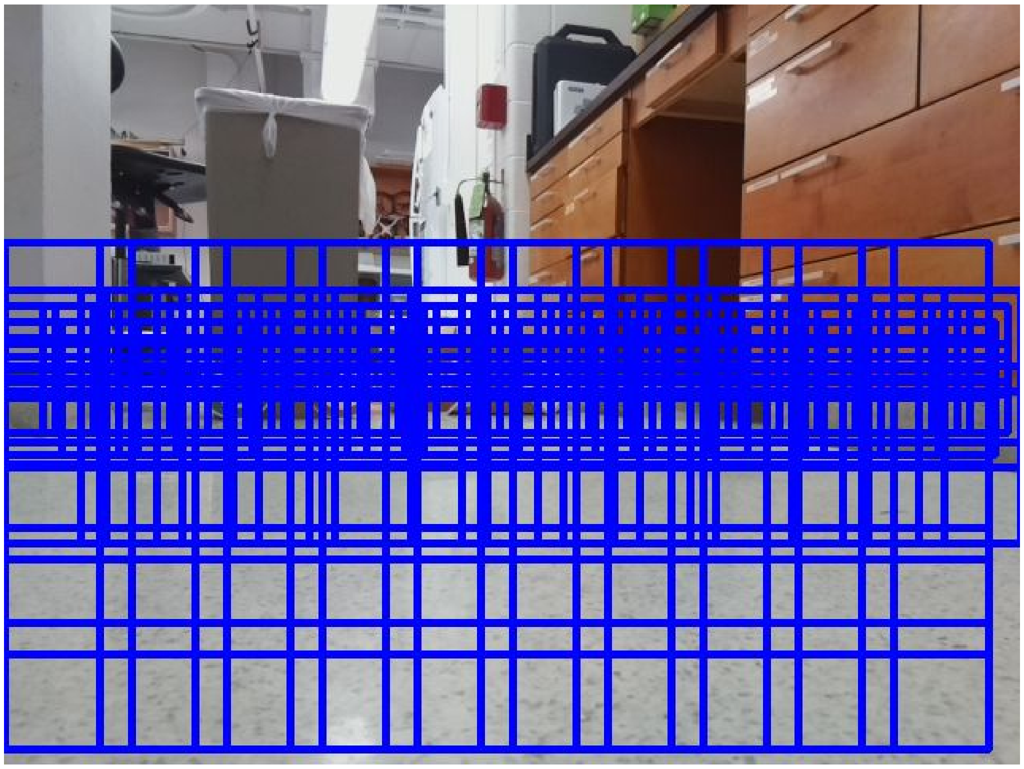


Figure 12. Multi-scale Sliding WIndow Search

4. Multiple Detections & False Positives

After implement multi-scale sliding window search based on our trained classifier, we get the following result:

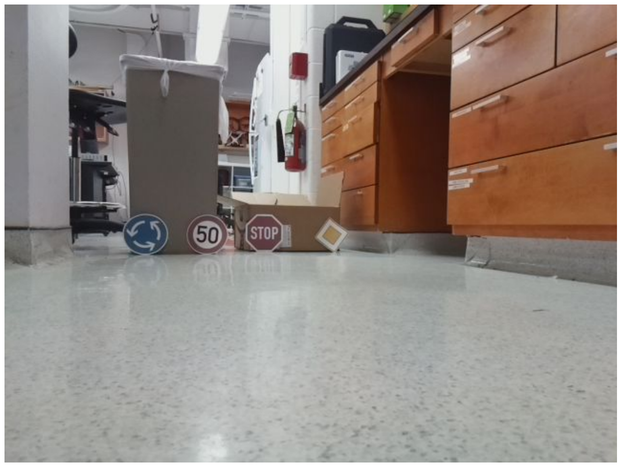


Figure 13. Original Image to Search

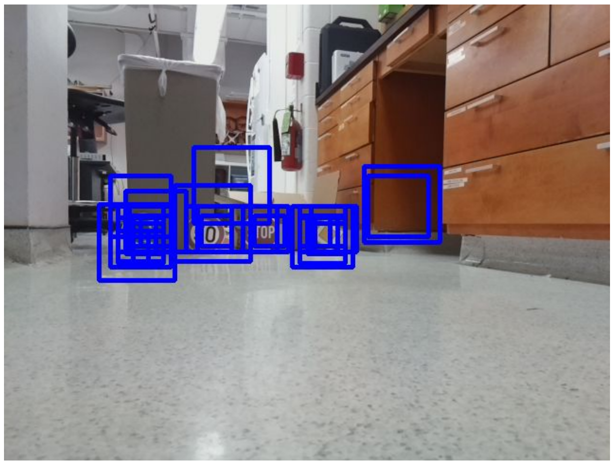


Figure 14. First Search Result of the Detector

As we can see, we faced with two problems:

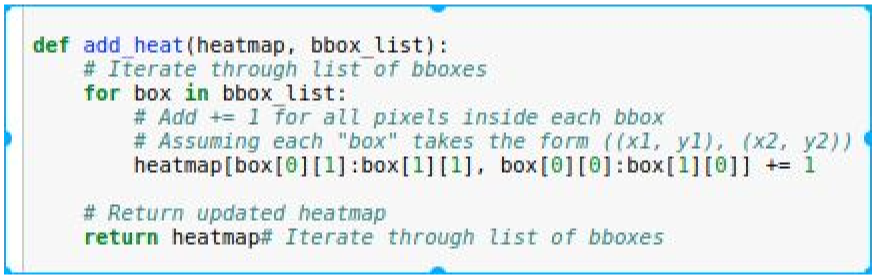
(a) Multiple positive: multiple windows overlaps on the same traffic sign

(b) False positive: windows appear on the non-traffic sign region

To overcome this two problems and make our detector more robust, I built a heat-map from these detections in order to combine overlapping detections and remove false positives.l

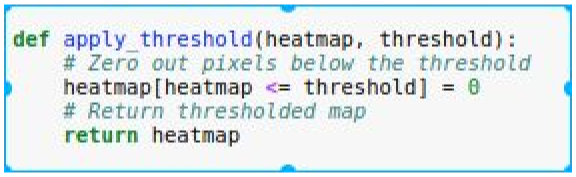
To make a heat-map, I simply add "heat" (+=1) for all pixels within windows where a positive detection is reported by the classifier.

I then wrote a function that adds "heat" to a map for a list of bonding boxes for the detections in images.



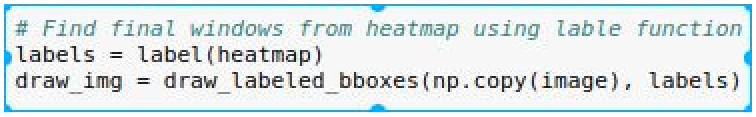
```python
def add_heat(heatmap, bbox_list):
    # Iterate through list of bboxes
    for box in bbox_list:
        # Add += 1 for all pixels inside each bbox
        # Assuming each "box" takes the form ((x1, y1), (x2, y2))
        heatmap[box[0][1]:box[1][1], box[0][0]:box[1][0]] += 1

    # Return updated heatmap
    return heatmap# Iterate through list of bboxes
```

To remove false positives, I wrote a function to reject areas by imposing a heat threshold:



```python
def apply_threshold(heatmap, threshold):
    # Zero out pixels below the threshold
    heatmap[heatmap <= threshold] = 0
    # Return thresholded map
    return heatmap
```

Finally, to figure out how many traffic signs in each video frame and which pixels belong to which traffic signs, my solution is to use the *lable()* function:



```python
# Find final windows from heatmap using lable function
labels = label(heatmap)
draw_img = draw_labeled_bboxes(np.copy(image), labels)
```

The finally output is as follow:

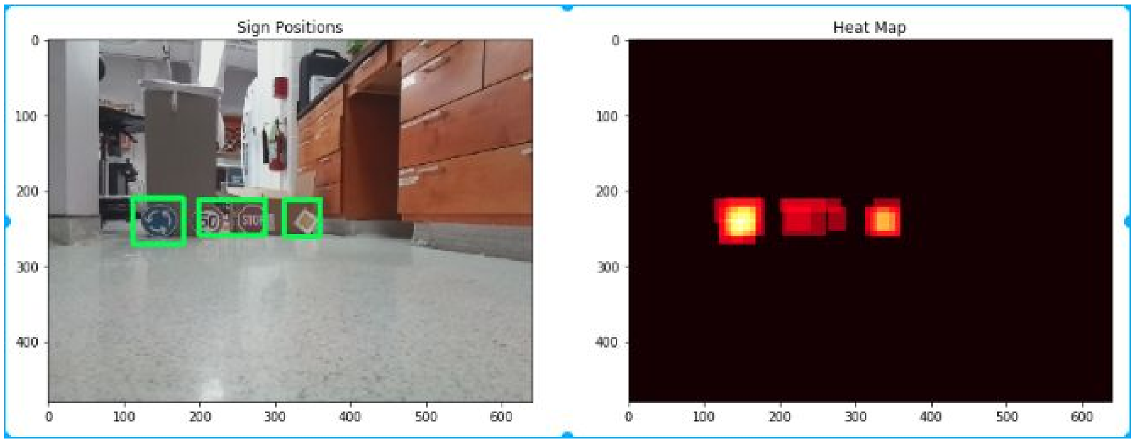


Figure 15. Final Search Result within Heat-map

As we can see, after apply the heatmap, we successfully remove all false positives and combined all multiple positives.(although the middle two traffic signs are combined due to there are nearby each other, we can separate them once the window are far not square.)

Part 3. Traffic Sign Classifier

This part, I build a deep CNN based on German Traffic Sign Dataset again, which can classifier 43 sort of traffic sign. It's accuracy arrived at 96.1%.

1. Dataset distribution and its preprocess

Images are 32(width)*32(height)*3(RGB channel)

Training data has 39209 images

Validation data has 2526 images

Test data has 10104 images

Dataset totally has 43 sorts(such as 20 km/h limitation, no passing, bumpy road etc.)

Before we feed the data into our model to train, we normalized the data, make it locate at the region [-0.5, +0.5], thus can shorten the training time highly.

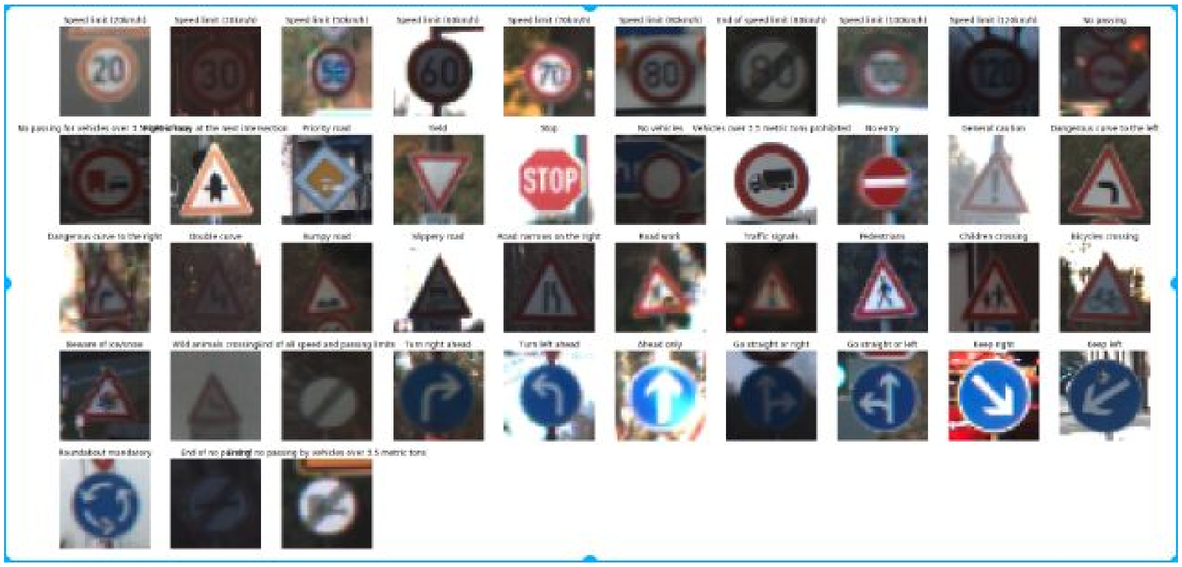


Figure 16. All Sorts of Traffic Sign of Dataset

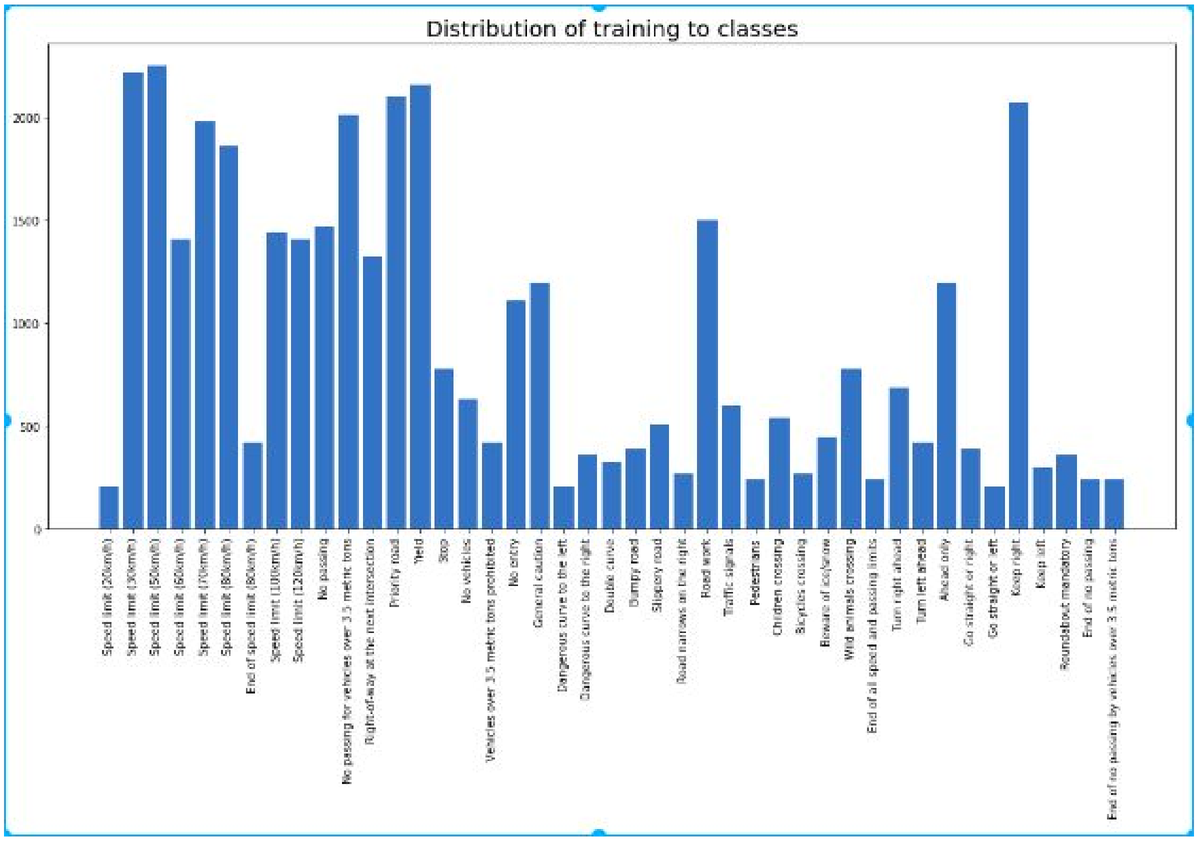


Figure 17. Unbalanced Distribution of Dataset

2. Deep CNN Architecture

As for the Convolutional Network structure, I refer Yann LeCun's this paper. The networks made up with 3 convolutional neural network, has 3*3 kernel, its depth of the next layer double, and has ReLU to serve as activation function. Each CNN has max 2*2 pooling. It has 3 layers of fully-connected layers, and generate 43 result at final layer.

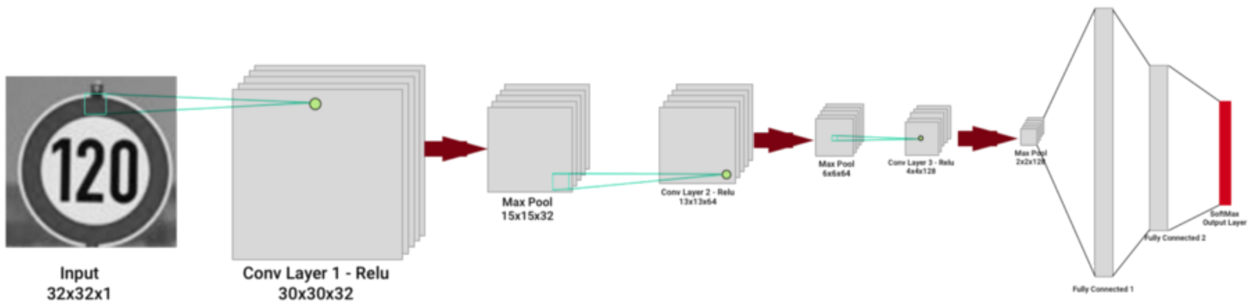


Figure 18. Deep CNN Architecture

3. Dropout

To improve the reliability of the model, I implement dropout algorithm. It can inhibit the model overfitting. According to this paper, the author choose different dropout value for different layers. And I decided to implement similar methods, so I define two dropout values: **p-conv** and **p-fc** one is for the CNN, the other is for the fully-connected layers. I choose a more active dropout value when it come to the end of the neural network, that because we don't want to lose too much information at the beginning of the network. I try several sets of dropout value, finally find **p-conv** = 0.7 and **p-fc** = 0.5 can make my model perform better.

4. Data Augment

As we can see in the dataset distribution, 43 sort of data has much unbalanced quantity. This can easily lead our model's prediction biased to certain sort of model. So I decided to implement data augment to balance the data. Moreover, to make the model more robust, I provide new image in different environment such as translation, rotation, transformation, blur and illuminance. I implement these effect respectively by *cv2.warpAffine*, *cv2.GaussianBlur* and *cv2.LUT*.

Some example of new data is as follow:

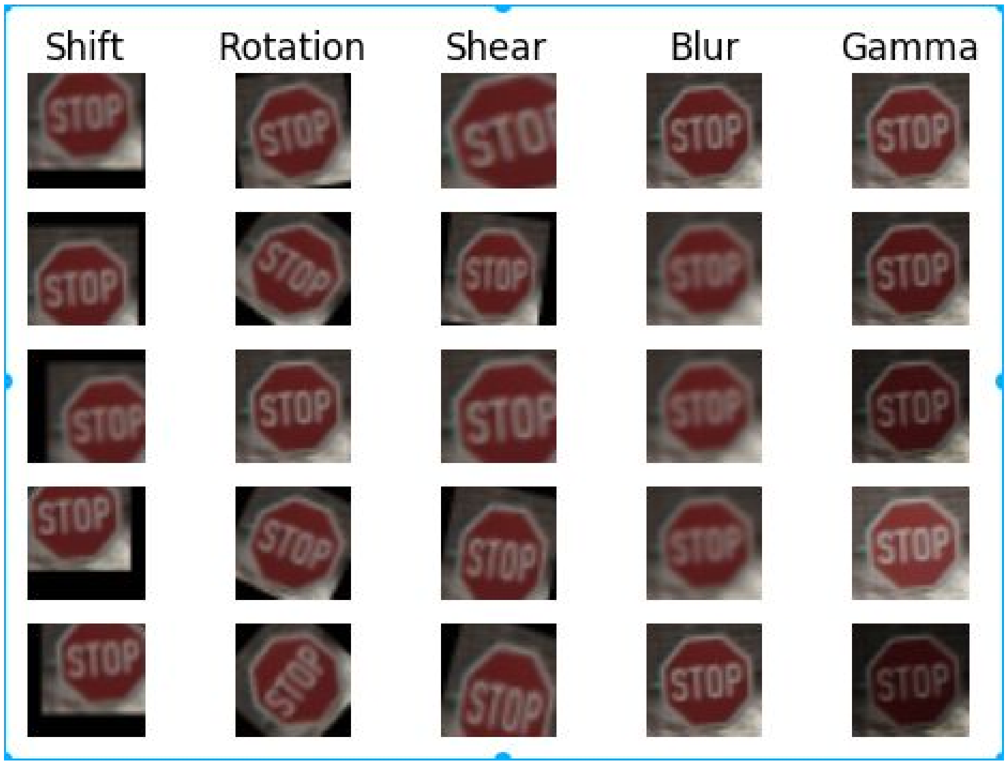


Figure 19. Effect of Data Augment in Different Ways

5. Model Performance

Here is the loss function value history through 100 training epochs.

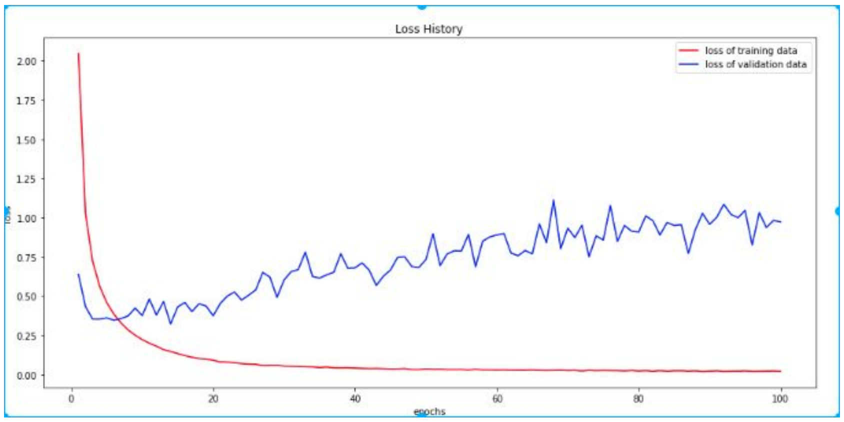


Figure 20. Loss function of training and validation dataset throughout the training

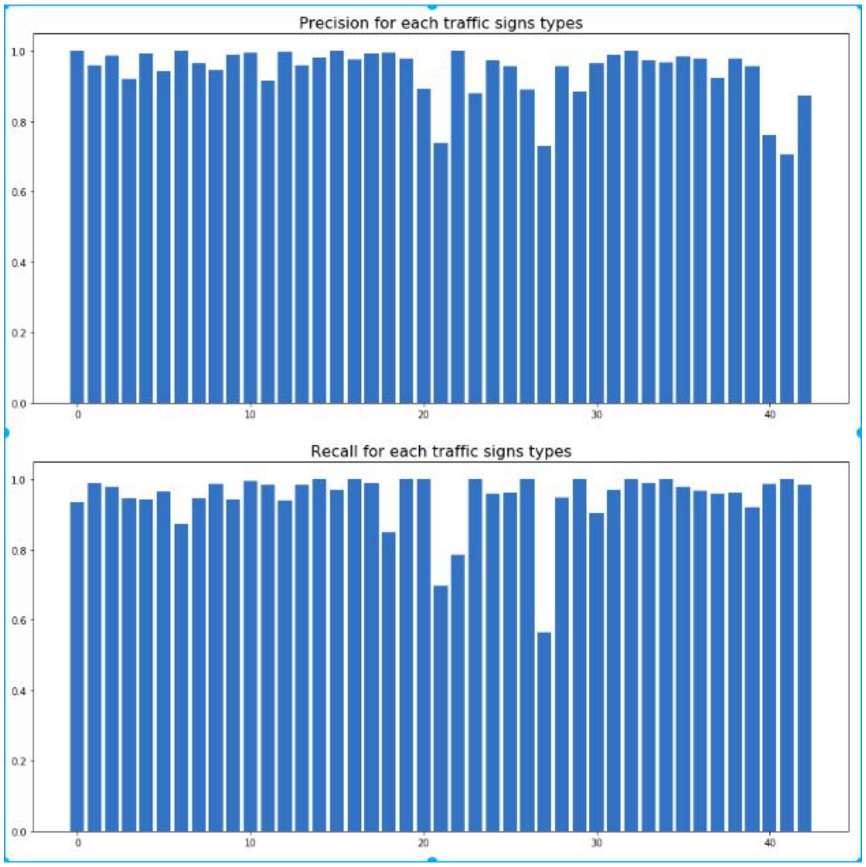


Figure 21. Precision & Recall for all of Traffic Sign's Sorts

From above result, we know that although we get good classifier performance(96.1 %), the validation loss rise slightly as the training loss decrease nearly to zero, which release that our model is a little overfitting.

In addition, the model did very badly for traffic sign 21 and traffic sign 27. This is because the deficiency of the original dataset and the lack of these two traffic sign images.

6. Performance of Traffic Sign Recognition (Detection & Classification)

After I fusion the traffic sign detection @ classifier together, here is two test image:

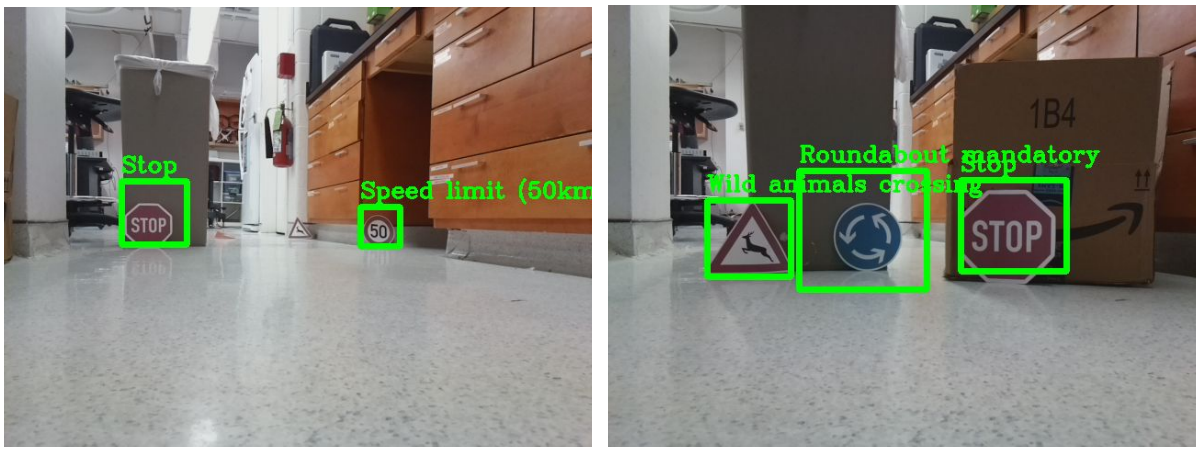


Figure 22. Final Output for Traffic SIgn Recognition

Stretch Goal

Now I can make the turtlebot3 robot to follow the Lane Lines automatically, and recognize the traffic sign he face stably. Specially, I programmed the robot to control its driving speed according to the speed limitation traffic sign. I'll continue explore more robot behavior combination based on traffic sign recognition. And my next goal is to realize **Visual SLAM** by monocular camera on the robot.